Factorisation of MaCh3 classes

MaCh3 Workshop

05.11.2026

Kamil Skwarczyński



Daniel Barrow

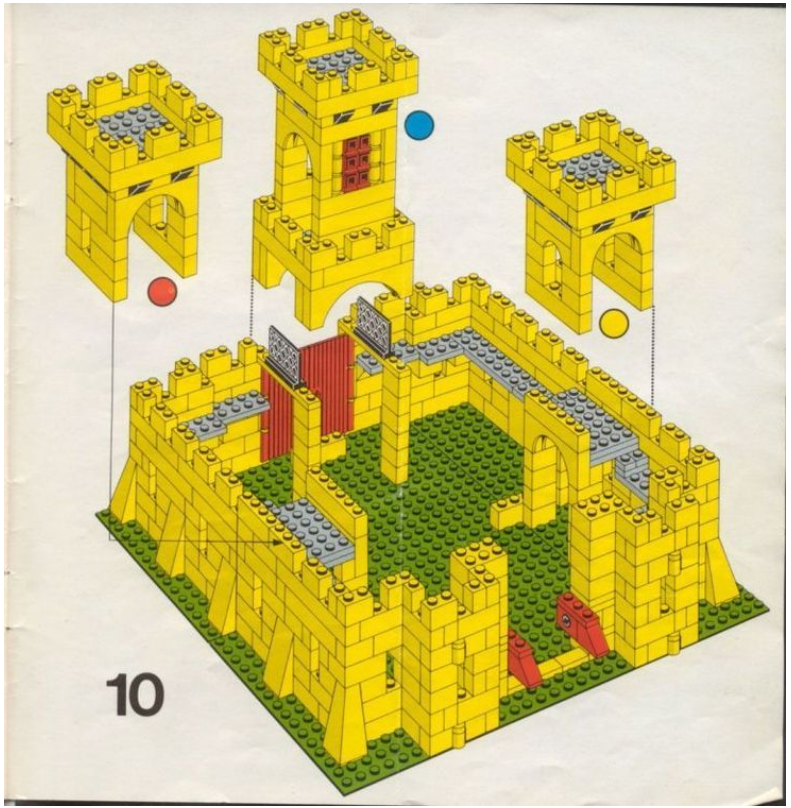


MaCh3 Lego

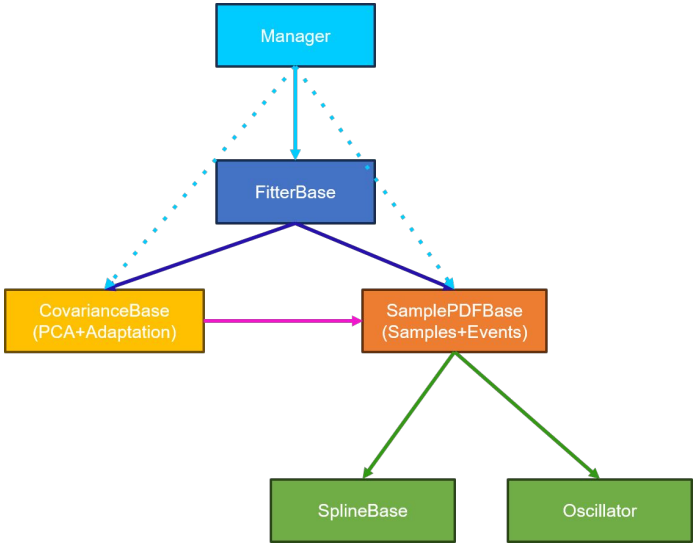
Goal of factorsiation is to have single class dedicated to a single task.

It allows developers/power users o more easily extend code by

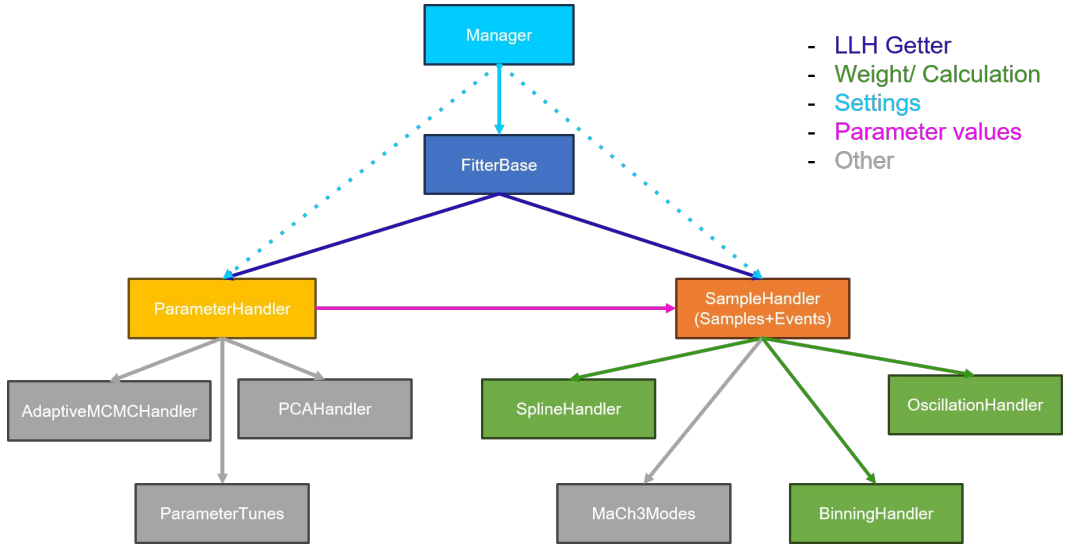
- adding new fitting algorithm etc (wacky MCMC)
- go beyond beam/atm oscillations
- go beyond neutrino (dark matter?)



Old Structure (~1.2.0)



Current



MaCh3 code-base over the years have become much more factorised and clean extracting functionality from behemoth classes into smaller more specialised one. Current structure still has quirks from initial design.

The aims of this talk is to identify:

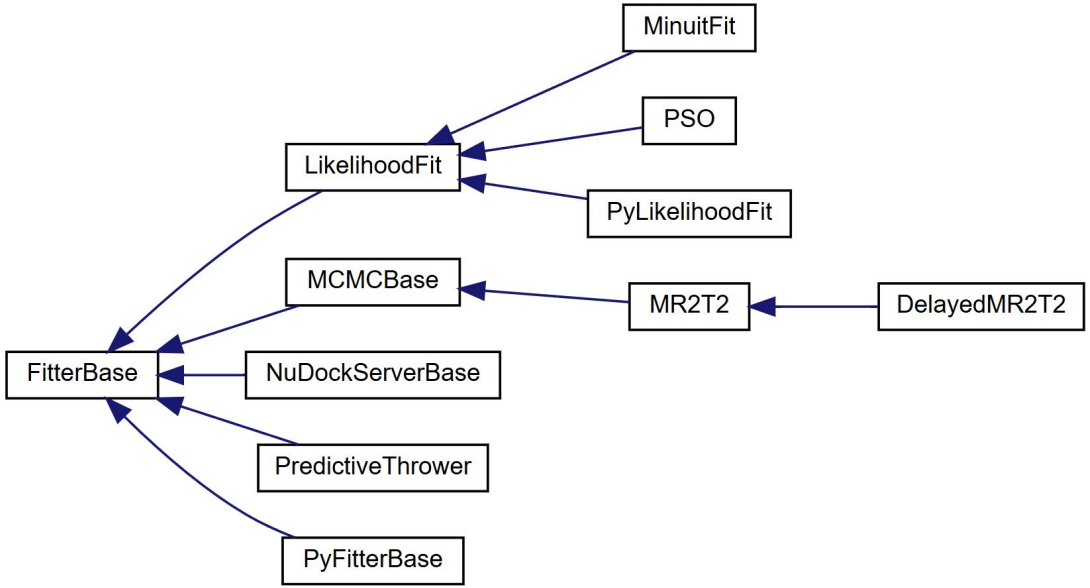
- where factorisation is sufficient
- where we should split further

Fitting Factorisation

Fitter Factorisation

Fitter Base allows to implement different MCMC algorithms, but also Likelihood-based minimisers as well as interface to external libraries.

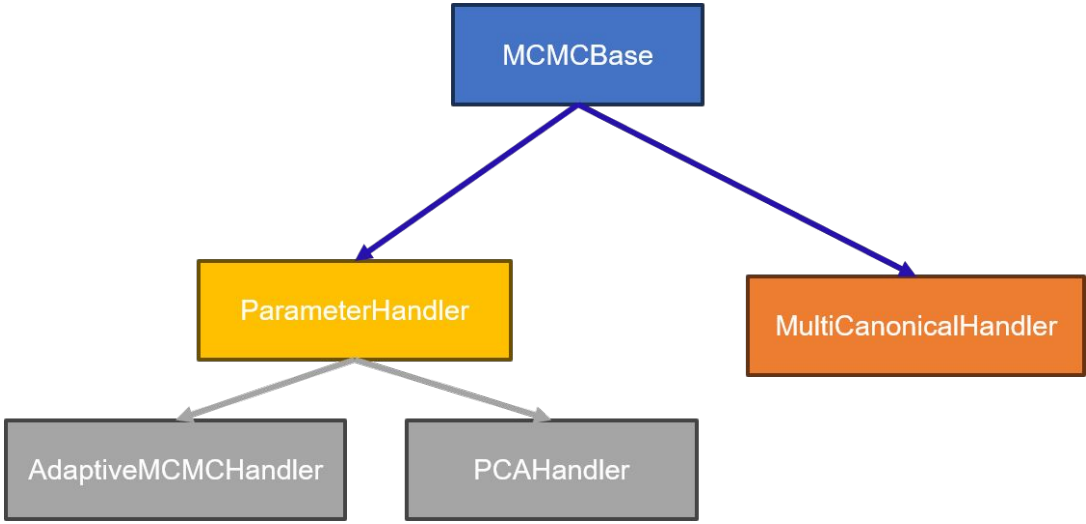
I will focus on MCMC Base as this is what concerns us most



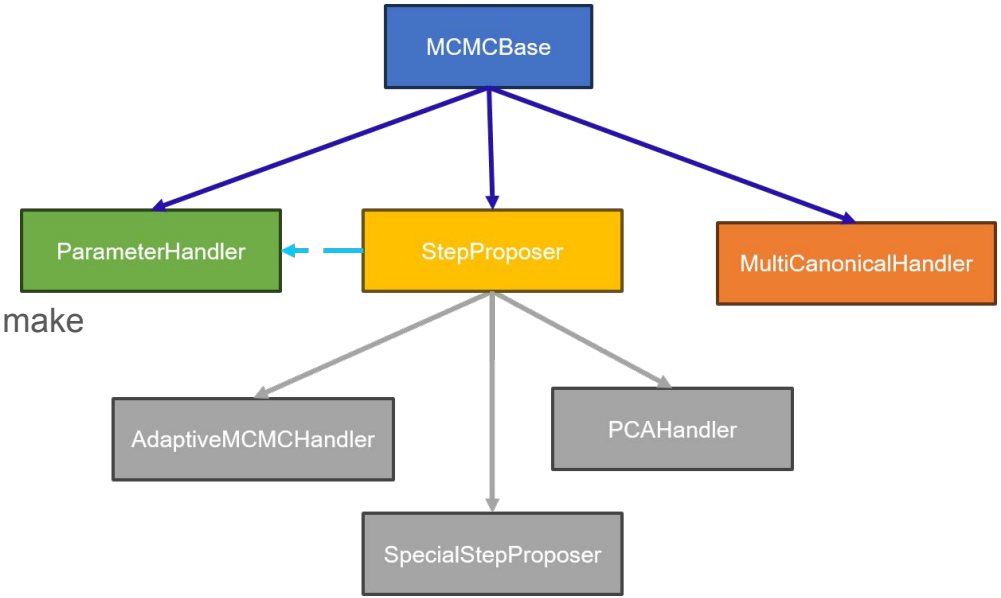
MCMC Base Factorisation

Step Proposal being in Parameter Handler is the main bottleneck when it comes being able to flexibly change step proposal.

Discussed before for example by Luke here



Example of Solution



By detaching Step Proposer from Param Handler it would make easier for MCMC Base to update proposing algorithm.

ParamHandler:

- ParamValues
- CovarianceMatrix/Priors
- LLH Calcualtions

StepProposer:

- Throw/ProposalMatrix
- PCA matrix ?
- Keep Pointer to ParamHandler(s) to override values
- StepScale ?

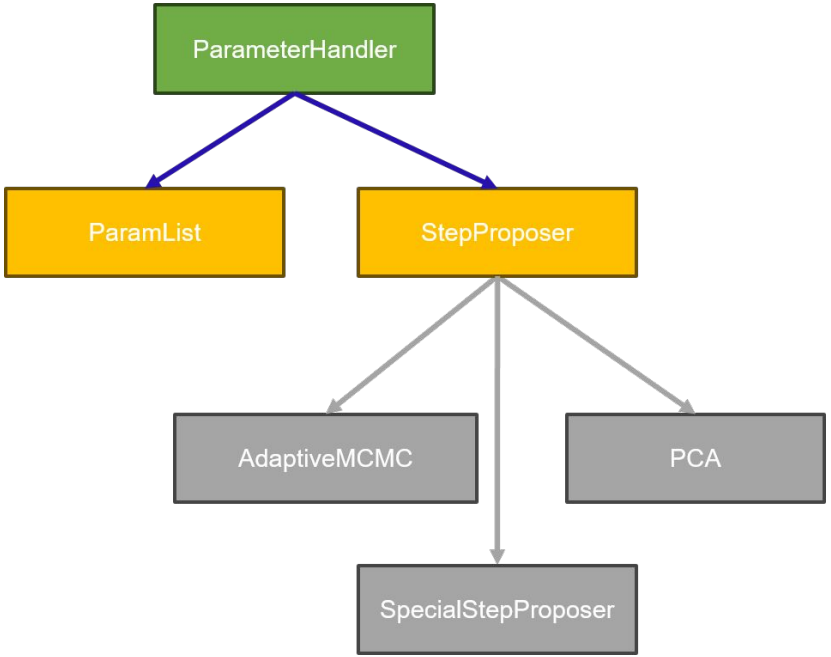
Step Proposer would be only used in MCMC Base and would be spawned by passing Param Handler

If we separate SpecialStepProposer for flips and other we could also easily control flips etc.

Example of Solution [Luke approach]

Alternative solution has been proposed by Luke.

There separation between param list and proposing is clear although everything under umbrella of ParamHandler rather than “MCMCBase”



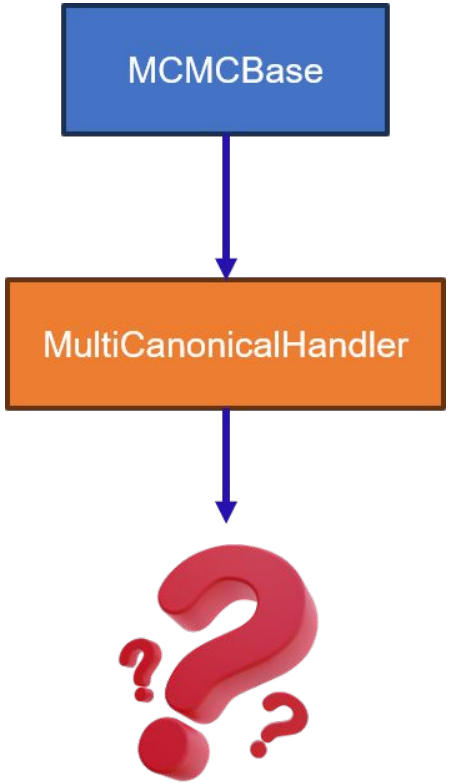
Moving Beyond

David Implemented Umbrella Sampling in nicely factorised way in MultiCanonicalHandler.

Liban demonstrated this class can be easily extended to use to allow tempered umbrella sampling

Hank implemented delta M2 prior penalty to allow DUNE make easier to calculate BF for mass ordering.

Do we want to convert MultCanonicalHandler into more generic “bias introducer”



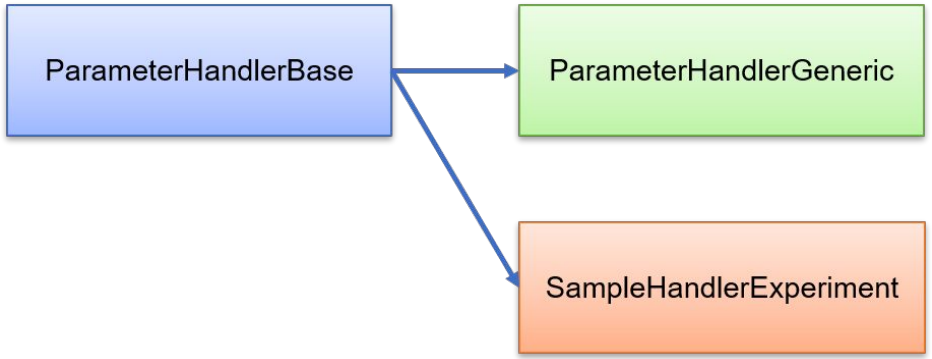
Parameter Factorisation

Param Handler

Currently, Param Handler give us freedom being able to provide whatever penalty we like.

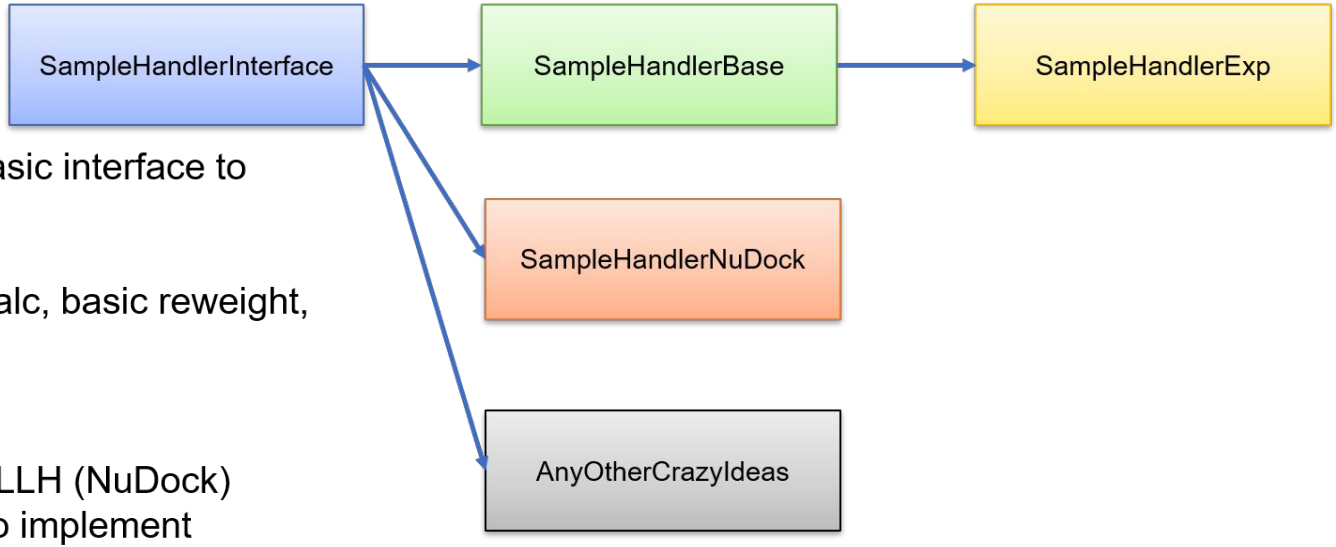
For example using chi2 histograms.

Overall factorisation is clean, we might consider whether we want move more functionality from **ParamGeneric** into **ParamBase** like **ParameterGroups**



Sample Factorisation

Current Sample Structure



SampleHandlerInterface – Basic interface to Fitting, posterior predictive etc.

SampleHandlerBase – LLH calc, basic reweight, uncertainty defintion etc.

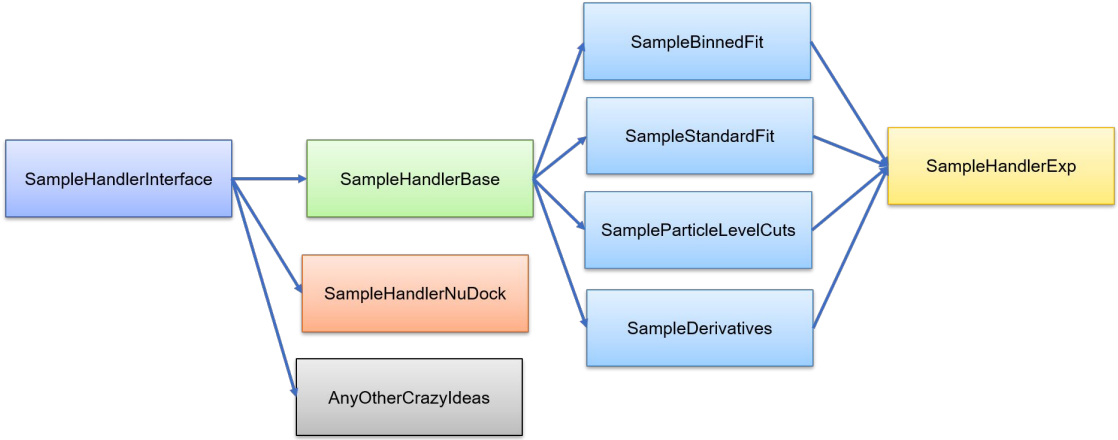
Advantages:

- Clear structure for external LLH (NuDock)
- Ability on experiment side to implement whatever they like

Problems:

- Right now plotting, selection development tools are bleeding into Base and Interface.
- Can't do sample migration
- Not clear how to handle fully binned analysis

Sample Factorisation



Within MaCh3 we have single reweighting routines with cuts and event migration.

Different users expressed different needs in the past:

- BinnedFitter - which will not need migration etc
- Particle Level cuts - give more flexibility but would induce speed hit to normal sample
- Derivatives calculations - useful for example for Hamiltonian MCMC

Having single class accommodating these needs is impossible, on the other hand we should ensure SampleHandlerBase shares as much as possible common code. Otherwise maintenance becomes difficult

This could be handled nicely like below:

SampleHandlerExp<SampleHandlerParticleLevel> SlowSample;

SampleHandlerExp<SampleHandlerStandardFit> NormalSample;

We should avoid diamond inheritance

Curiously Recurring Template Pattern (CRTP)

We have ExperimentSpecyfic EventInfo and EventInfo in Core.

They have 1-to-1 mapping, i.e. describe same event. However, they are separate objects.

CRTP allows mitigating this by making data class/struct template.

SampleHandlerBase would know about experiment class.

Cleaner code and potneitally faster code -> avoid memory jumping from one object to another as we loop over events

```cpp
/// @brief struct holding basic SK info necessary for reweighting
struct SKEventInfo {
  /// Pointer to SK detector matrix bin
  const double* skdet_pointer = &(M3::Unity);

  /// Includes POT, flux and all other weights which are constant
  /// @note YSP: Nominal xsec weight and Radiative correction weights for T2K-MC 23c
  double Nominal_w = 0;

  /// Value of Coulomb Correction
  double CCCor = 0;

  /// Oscillation channel code used by T2K-SK, doesn't have to match one used by MaCh3
  int OscChannel = 0;
  /// NEUT Neutrino interaction mode
  int neut_mode = 0;
  /// Whether the interaction is neutral current
  bool isNC = 0;
  /// PDG code of final state neutrino
  int nupdg = 0;
  /// PDG code of unoscillated neutrino
```

```cpp
template<typename EventDataType>
SampleHandlerBase_t<EventDataType>::SampleHandlerBase_t(
                                    std::string ConfigFileName,
                                    ParameterHandlerGeneric* xsec_cov,
                                    const std::shared_ptr<OscillationHandler>& OscillatorObj_)
                                    : SampleHandlerInterface() {
```

Sample Factorisation - Plotting

Currently, Get2DVarHist and associated plotting tools separately treat “event info” and “particle info”. This introduces an explicit distinction between “event” and “subevent” throughout the codebase

From an API POV the separation is between single-valued and multi-valued projections of a 'row' (usually event)

We could use C++ templates function which handles single value and vectors.

The goal is to eliminate the event/subevent distinction from the plotting API and make “Interface” function signatures clearer

```
std::unique_ptr<TH2> Get2DVarHist(const int iSample,
                                  const std::string& ProjectionVarX,
                                  const std::string& ProjectionVarY,
                                  const std::vector< KinematicCut >& EventSelectionVec = {},
                                  int WeightStyle = 0,
                                  const std::vector< KinematicCut >& SubEventSelectionVec = {})
                                  final;
```

```
// ***********************************************
std::unique_ptr<TH2> SampleHandlerBase::Get2DVarHist(const int iSample,
                                                     const std::string& ProjectionVar_StrX,
                                                     const std::string& ProjectionVar_StrY,
                                                     const std::vector< KinematicCut >& EventSelectionVec,
                                                     const int WeightStyle,
                                                     const std::vector< KinematicCut >& SubEventSelectionVec) {
// ***********************************************
  //DB Need to overwrite the Selection member variable so that IsEventSelected function operates correctly.
  //   Consequently, store the selection cuts already saved in the sample, overwrite the Selection variable, then reset
  auto tmp_Selection = ApplyTemporarySelection(iSample, EventSelectionVec);

  //DB Define the histogram which will be returned
  //KS: If we use 2D non uniform binning and wanting to plot fit variables use TH2Poly everything else uses TH2D with binning from config
  std::unique_ptr<TH2> _h2DVar;
  if(GetNDim(iSample) == 2 && !Binning->IsUniform(iSample) &&
     ProjectionVar_StrX == GetKinVarName(iSample, 0) && ProjectionVar_StrY == GetKinVarName(iSample, 1) ) {
    _h2DVar = std::unique_ptr<TH2>(static_cast<TH2*>(M3::Clone(GetMCHist(iSample)).release()));
  } else {
    std::vector<double> xBinEdges = ReturnKinematicParameterBinning(iSample, ProjectionVar_StrX);
    std::vector<double> yBinEdges = ReturnKinematicParameterBinning(iSample, ProjectionVar_StrY);
    _h2DVar = std::make_unique<TH2D>("", "", int(xBinEdges.size())-1, xBinEdges.data(), int(yBinEdges.size())-1, yBinEdges.data());
  }
  _h2DVar->SetDirectory(nullptr);
  _h2DVar->GetXaxis()->SetTitle(ProjectionVar_StrX.c_str());
  _h2DVar->GetYaxis()->SetTitle(ProjectionVar_StrY.c_str());
  _h2DVar->GetZaxis()->SetTitle("Events");

  bool IsSubEventHist = IsSubEventVarString(ProjectionVar_StrX) || IsSubEventVarString(ProjectionVar_StrY);
  if (IsSubEventHist) Fill2DSubEventHist(iSample, _h2DVar.get(), ProjectionVar_StrX, ProjectionVar_StrY, SubEventSelectionVec, WeightStyle);
  else {
    //DB Grab the associated enum with the argument string
    int ProjectionVar_IntX = ReturnKinematicParameterFromString(ProjectionVar_StrX);
    int ProjectionVar_IntY = ReturnKinematicParameterFromString(ProjectionVar_StrY);

    //DB Loop over all events
    for (unsigned int iEvent = 0; iEvent < GetNEvents(); iEvent++) {
      const int EventSample = MCEvents[iEvent].NominalSample;
      if(EventSample != iSample) continue;
      if (IsEventSelected(EventSample, iEvent)) {
        double Weight = GetEventWeight(iEvent);
        if (WeightStyle == 1) {
          Weight = 1.;
        }
        double VarX = ReturnKinematicParameter(ProjectionVar_IntX, iEvent);
        double VarY = ReturnKinematicParameter(ProjectionVar_IntY, iEvent);
        _h2DVar->Fill(VarX,VarY,Weight);
      }
```

Questions

- How to allow more reweighting routines [binned, particle level shifts, standard etc]
- How to abstract slow vs fast to calculate variables so they can be used in plotting and fitting
  - More options to plotting breakdown?
- How to add more freedom to fit and step proposal (detach step proposal from Param Handler?)
- Should MaCh3 move to CRTP?
- Should we move features from ParamHandlerGeneric to ParamHandlerBase?
  - ParamGroup?
  - What hooks should ParamBase have? (Look at T2K+SK)
- Is factorisation sufficient for MaCh3:
  - beyond Beam/Atm like Solar/Reactor
  - beyond oscillation analysis (Xsec) or n
  - beyond neutrino physics (Collider, dark matter)
  - differentiable reweighting (for Hamiltonian MCMC)

Bonus:

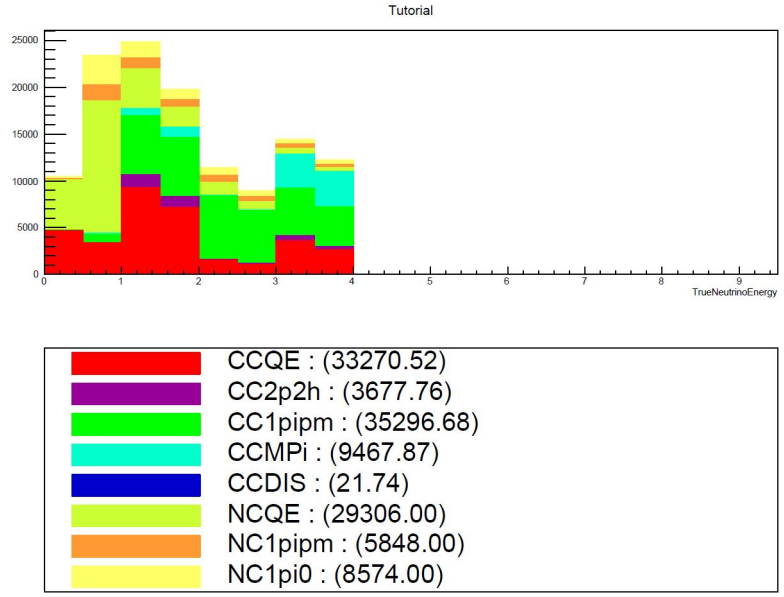
- Is current factorisation good for Parallel Tempering (MPi) or other algorithms?
- Is factorisation fine to implement N-dimensional splines/grid calculations or other systematic evaluation methods
- Is factorisation sufficient for external fitting/sampling libraries (like SBI) that we would like to use via pyMaCh3/NuDock

Plotting Breakdown

We have ability to produce breakdown by mode and osc channel.

Would be good to extend into target, nu pdg etc.

Can we do this fully customizable for user?




```
// ************************************************
std::vector<std::unique_ptr<TH2>> SampleHandlerBase::ReturnHistsBySelection2D
const std::string& KinematicProjectionX,



// ************************************************
  std::vector<std::unique_ptr<TH2>> hHistList;

  const int iMax = GetRangeForPlotType(Selection1, iSample);
  for (int i=0;i<iMax;i++) {
    if (Selection1 == SamplePlotType::kModePlot) {
      hHistList.push_back(Get2DVarHistByModeAndChannel(iSample,
      KinematicProjectionX,KinematicProjectionY,i,Selection2,WeightStyle));
    }
    if (Selection1 == SamplePlotType::kOscChannelPlot) {
      hHistList.push_back(Get2DVarHistByModeAndChannel(iSample,
      KinematicProjectionX,KinematicProjectionY,Selection2,i,WeightStyle));
    }
  }

  return hHistList;
}
```